(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
15 February 2001 (15.02.2001)

PCT

(10) International Publication Number
WO 01/11471 A1

(51) International Patent Classification[7]: G06F 13/00

(21) International Application Number: PCT/US00/21791

(22) International Filing Date: 9 August 2000 (09.08.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/148,090 10 August 1999 (10.08.1999) US

(71) Applicant *(for all designated States except US)*: **OZ.COM** [US/US]; 77 South Bedford Street, Burlington, MA 01803 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **ÞORVALDSSON, Haraldur, D.** [IS/IS]; Hjardarhagi 32, IS-107 Reykjavík (IS). **EMILSSON, Kjartan, Pierre** [IS/IS]; Sörlaskjól 22, IS-107 Reykjavík (IS). **GUDJÓNSSON, Gudjón** [IS/IS]; Brúnastadir 77, IS-107 Reykjavík (IS).

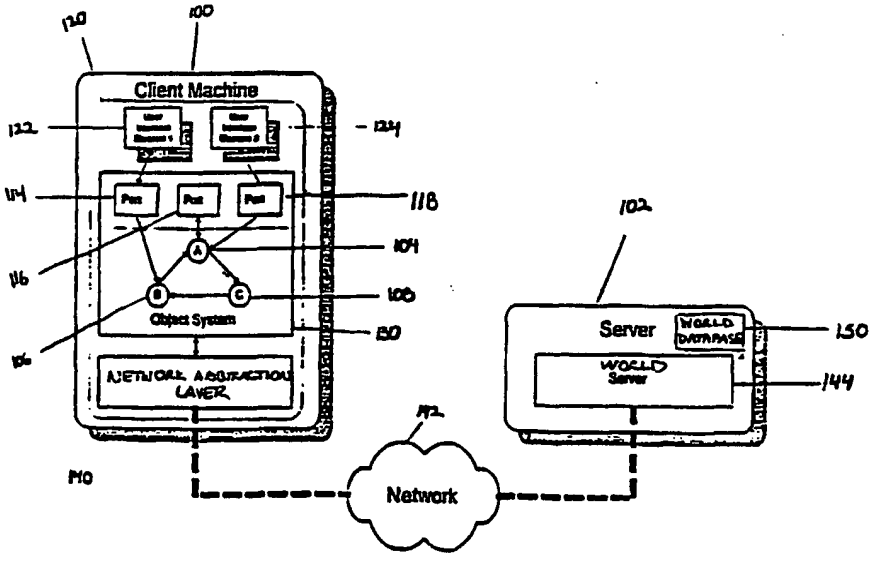
(74) Agent: **NOAH, Todd**; Dergosits & Noah LLP, Four Embarcadero Center, Suite 1150, San Francisco, CA 94111 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR DISTRIBUTED MULTI-USER REAL-TIME SIMULATIONS OVER AN ELECTRONIC NETWORK



(57) Abstract: An object oriented method and system is provided for facilitating the creation of distributed real-time simulations. At least one remote server (102) running a World Server application (144) hosts one or more worlds. Each world contains any number of nodes (104, 106, 108). A connected client (108) running a client software application (120) receives current values of node variables, can change variable values, can create nodes and can destroy nodes for the world. All such changes are persistently recorded by the server (102) in the World Database (150). A client (108) subscribes to an active set of a world's nodes. The server (102) sends the client (108) updates in real-time for any values of variables in the client's active set changed by other clients connected to the same world. The Object System (130) handles general chores of replicating worlds across clients and ensuring that node behavior initiated on one client is replicated on other clients that have that node in their active sets. The Object System (130) is responsible for synchronizing nodes that are running on different clients so that each node appears to all participants to have a single identity and state Object System compatible components provide

WO 01/11471 A1

patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:

— *With international search report.*

— *Before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

world/domain specific behavior. A remote World Server (150) handles lower-level activities of persistently storing and distributing world data. Node behavior is executed on clients only. Ports (114, 116, 118) are used to channel input and output to and from nodes in the world. Ports are non-shared and handle computer-oriented processes and elements, while nodes are shared and deal with

SYSTEM AND METHOD FOR DISTRIBUTED MULTI-USER REAL-TIME SIMULATIONS OVER AN ELECTRONIC NETWORK

BACKGROUND OF THE INVENTION

1. Field of the Invention:

The present invention relates generally to virtual reality and, more particularly, to the creation and distribution of multi-user real-time simulations over an electronic network.

2. Description of Related Art:

With the increase in computer technology, the creation of virtual worlds that are shared over an electronic network, such as the Internet or a Local Area Network has become feasible. Complex virtual worlds require potentially hundreds or thousands of data objects. Each user of the shared virtual world can make changes to these data objects. To maintain a common shared experience, all such changes must be replicated to each user of the shared virtual world.

The replication process according to the prior art is subject to several limitations. First, the server computer that stores the shared world simulation must continually update its shared world data without disturbing the shared experience. This requires that the server computer have an extremely large data storage capacity and a powerful processor. In addition, each data change must be transmitted to all users of the shared world. Thus, each user of the shared virtual world must have sufficient bandwidth to continually receive updates without significantly affecting the virtual world. Furthermore, each user must also have sufficient storage capacity and processing power to receive this updated information while maintaining the virtual experience. Additionally, many subsystems of a typical client employed by a user to participate in the virtual world, such as graphical renderers or physics simulators, can only handle limited data transfer before becoming overwhelmed. These prior art limitations in data storage, processing power, and bandwidth significantly restrict the practical size and complexity of a shared virtual world.

One solution to these problems has been to enable clients to only download and run a subset of the world at any given time. Many prior art distributed virtual reality systems partition the world along spatial boundaries, such as the rooms of a building or sections of a battlefield. However, such spatially-bound schemes are not very flexible and make the assumption that objects must always be near each other in space to interact. This assumption does not apply to many actions that can be performed in a shared virtual world, such as a virtual telephone conversation.

It would therefore be an advantage to provide a shared world scheme that is readily updated without disturbing the shared virtual experience. It would be a further advantage if updating and maintaining this shared world scheme required reduced storage and processor capabilities than the prior. It would be yet another advantage if maintenance of this shared world were not subject to user bandwidth limitations.

SUMMARY OF THE INVENTION

The present invention is an object oriented method and system for facilitating the creation of distributed real-time simulations. The system comprises a server or a plurality of servers, each running the system's server software, and one or more clients, each running the system's client software. Each server can host one or more world simulation databases, called worlds. A world contains any number of data objects called *"nodes."* Each node has one or more data attributes, or *"variables"* as well as references to other nodes.

A client that is connected to the server receives the current values of the variables of nodes, can change the values of variables, can create nodes and can destroy nodes for the particular world. All such changes are persistently recorded by the server in the world database.

In the present invention, a client subscribes to a subset of a world's nodes. This set is called the client's *"active set"* of nodes. The client can also unsubscribe to nodes, so the active set of a client can vary over the duration of a connection with a server and a world. The server sends update notifications to a client in real-time when the values of variables in the client's active set are changed by other clients connected to the same world.

In the present invention, the Object System is a core component of the shared-world client software application. The Object System according to the present invention handles the

general chores of replicating worlds across clients while Object System compatible components provide world/domain specific behavior. The types of a world are not specified by the Object System but are defined and installed into the object system as modular components, without changes to the Object System's software. The displaying and rendering of spaces and dimensions is performed independently of the Object System. All nodes in an Object System world belong to a "*node type*" that specifies the type of attributes a node has. A node type can optionally have executable code associated with it, referred to herein as the node's *"behavior"*. The structure of the installed node types and their behavior determines the structure and semantics of the world's simulation so the Object System can be quickly adapted to many kinds of simulated worlds. Node behavior is executed on clients only, not on the server. The Object System ensures that node behavior initiated on one client is replicated on other clients that have that node in their active sets. The Object System is responsible for synchronizing nodes that are running on different clients, so that each node appears to all participants to have a single identity and state

The Object System specifies a set of interfaces that define the communication between the Object System and nodes. These interfaces are designed such that nodes view data abstractly and do not call network functions directly. A central design feature of the preferred embodiment is that a node does not know at runtime whether it is executing as a consequence of an event originating on its local client or whether it is replicating behavior that originated on a remote client. Nodes belong to the world and are *"shared."*

The Object System also defines interfaces which are implemented by plug-in components referred to herein as *"ports."* These ports are input/output components that are used to channel user input to and output from nodes in the world simulation. Ports are generally used to handle user interface ("UI") processes and elements, while nodes deal with processes and elements that exist in the simulated world. Ports belong to the client on which they run and are non-shared.

The amount of data sent to a client is determined by the size of the client's active set rather than by the overall size of the world. Because the client can subscribe to an arbitrarily small part of the world at any one time, it becomes feasible using the present invention for the server to create very large worlds. In addition, because the node behavior is executed by the clients, the workload on the server is less than if it had to execute all behavior on behalf of all

clients. Therefore, a single server can serve more clients at the same time than would otherwise be possible. Also, because each client subscribes only to a limited part of the world, the server does not have to replicate all behavior initiated by all other clients in the world, but rather only the behavior initiated by other clients on nodes in the client's active set.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a system diagram of the basic structure of a typical Object System client according to the present invention.

Figure 2 is an example of a state update cascade according to the present invention.

Figures 3a-c illustrate changes to an active set according to the present invention.

Figure 4 is a diagram showing an example of object and interfaces in a client according to the present invention.

Figure 5 is a flow chart of the initialization phase of a new node instance according to the present invention.

Figure 6 is a diagram of an exemplary simple Object System client according to the present invention.

Figure 7 is a diagram of an exemplary web-browser embedded client according to the present invention.

Figure 8 is a diagram of the graphic node and port types in an exemplary shared whiteboard according to the present invention.

Figure 9 is a diagram of the ports and nodes of an exemplary simplified 3D client according to the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Overview

The present invention is an object oriented method and system for facilitating the creation of distributed real-time simulations. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be evident, however, to one skilled in the art that the present invention may be practiced without the specific details. In other instances, well-known structures and devices are shown in block diagram form to facilitate explanation. The description of preferred embodiments is not intended to limit the scope of the claims appended hereto.

The present invention can be used for purposes including but not limited to creating virtual business environments, conferencing, planning, and electronic gaming. In the preferred embodiment, the present invention is implemented using a plurality of computers. Such computer can include but is not limited to a personal computer, network computer, network server computer, dummy terminal, local area network, wide area network, personal digital assistant, work station, minicomputer, and mainframe computer. The system according to the preferred embodiment of the present invention is a client-server system comprising a server or a plurality of servers for hosting one or more simulation databases, and a client or plurality of clients for accessing the simulation database(s). One skilled in the art will recognize, however, that the invention can also be implemented as any networked system comprising a plurality of computers.

In the preferred embodiment, the server computer is connected to the client computer through an electronic network such as the Internet, a Wide Area Network (WAN), a local area network ("LAN") or any combination thereof. This electronic network is implemented using any well-known hardware and software components. In the preferred embodiment of the invention, data is transmitted across the electronic network any appropriate communication device or connection, including but not limited to modem, satellite transmission, cable transmission, Ethernet, ADSL, ISDN, or T1.

The present invention can be used with any other hardware components that facilitate the creation and use of the distributed simulated environment. Such hardware components include

but are not limited to video capture boards, sound cards, rendering devices, sound recording equipment, film editing equipment, video cameras, joysticks, game controllers, graphics tablets, and microphones. In addition, the features of creating, maintaining, and using the distributed simulated environment can be implemented as one or more software applications, software modules, firmware such as a programmable ROM or EEPROM, hardware such as an application-specific integrated circuit ("ASIC"), or any combination of the above. Furthermore, the present invention can be used with any appropriate third-party software applications, for example, animation rendering applications, music composition applications, sound mixing applications, and graphics applications.

For example, one or more three-dimensional rendering software applications accessible to a transmitting network client computer can be used to render an object, such as an avatar, that can be incorporated into the shared simulated environment. A music composition application can be used to generate a musical soundtrack to accompany an action performed in the simulated environment. Any or all of the software applications or hardware configurations of the present invention can be implemented by one skilled in the art using well known programming techniques and hardware components.

General Description

The system according to the present invention comprises a server or a plurality of servers and plurality of clients (a client-server system). Each server runs the system's server software, while each client runs the system's client software. Each server can host one or more simulation databases, called world databases (*"worlds"*). A world contains any number of data objects called *"nodes"*. Each node has one or more data attributes, called *"variables."* Nodes can also have references to other nodes. Thus, the nodes in a world form a directed graph of interconnected nodes. In the present invention, nodes and references are sufficient to model most or all of the "reality" captured in a virtual world.

To connect to a world according to the preferred embodiment of the invention, the client must connect to the server and provide a valid username/password pair that has previously been registered with that server. If the username/password pair is validated by the server, the client is permitted to specify the world to which the client wishes to connect. A connected client can receive from the server the current values of the variables of nodes, can change the values of

variables, can create nodes and can destroy nodes for the particular world. All such changes are persistently recorded by the server in the world database, which is accessible thereto. The server then sends update notifications to a client in real-time when the values of variables are changed by other clients connected to the same world.

While it is possible to send a client data updates for all nodes and variables in a world, in the preferred embodiment, a client subscribes to a subset of a world's nodes. This set is called the client's *"active set"* of nodes. In this embodiment, therefore, a client only receives data updates for nodes in its active set.

Because a client may not know in advance which nodes exist in the selected world, an algorithm (*"active set traversal"*) is executed on the server to help a client select its active set. A client initially subscribes to one or more nodes. The algorithm is then used by the server to calculate which additional nodes are needed by the client and automatically adds these nodes to the client's subscription. As a result, the amount of data sent to a client is not determined by the overall size of the world but, rather, by the size of the client's active set. It therefore becomes feasible for the server to create very large worlds because the client can subscribe to an arbitrarily small part of the world at any one time. The client can also unsubscribe to nodes, so the active set of a client can vary over the duration of a connection with a server and a world.

Each variable has a textual name and a description of what type of value it stores, such as number, text, etc. The set of variables a particular node has is specified at node creation time and does not change afterwards. A set of named and typed variables comprises a *"node type."* There is a limited number of node types in the world, and each node is of exactly one type. The type of a particular instance of a node is specified at the node's creation time and cannot be changed afterwards. The terms type and interface will be used interchangeably herein.

In the present invention, the Object System is a core component of the shared-world software program (client). It is a software framework for building simulated (virtual) worlds within which geographically separated but networked computer users can experience and interact while being presented with the impression that there is only a single, mutually shared world. The world is persistently stored, so users can enter and leave the world without shattering this impression.

The Object System according to the present invention handles the general chores of replicating worlds across clients while Object System compatible components (plug-ins) provide

world/domain specific behavior. Therefore, it is not necessary to continually re-invent low-level data distribution protocols. As a result, more system and programmer resources can be devoted to creating functionality and behavior in the world domain.

The types of a world are not specified by the Object System but are defined and installed into the Object System as modular components, without changes to the Object System's software. A node type can (and usually does) have executable code associated with it, referred to herein as the node's *"behavior"*. It is in fact the structure of the installed node types and their behavior that determines the structure and semantics of the world's simulation, not the object system, so the Object System can be quickly adapted to many kinds of simulated worlds, without changes to the Object System's software.

The execution of node behavior code is triggered by a change in some of the node's variables. Node behavior is executed on clients only, not on the server. The Object System according to the present invention ensures that node behavior initiated on one client is replicated on other clients that have that node in their active sets. This way, the Object System gives the clients with the convincing illusion that they are in fact operating on the same, shared copy of the world, because each client immediately sees and experiences data changes and behavior initiated by the other clients, providing a distributed multi-user real-time simulation.

Because the node behavior is executed by the clients, the workload on the server is less than if it had to execute all behavior on behalf of all clients. Therefore, a single server can serve more clients at the same time than would otherwise be possible. Also, because each client subscribes only to a limited part of the world (its active set), the server does not have to replicate all behavior initiated by all other clients in the world, but rather only the behavior initiated by other clients on nodes in the client's active set. This, again, increases scalability.

The behavior code of nodes can be written by programmers using programming interfaces exposed by the object system. As a result, a node programmer does not have to be informed of the underlying network protocol of the Object System or to deal with the complexities of correctly replicating and making persistent data changes and node behavior. The Object System according to the present invention makes programming of distributed simulations resemble that of programming non-distributed simulations. As a result, distributed shared simulations can be programmed faster and at a lower cost.

The Object System

The Object System is a component of the client software application that is configured to handle general tasks of replicating worlds across clients. Object System compatible components are used to provide world/domain specific behavior. Because these components connect to form (directed) graphs, as will be explained later, they will be referred to herein as *"nodes."* The present invention facilitates the development of nodes, thereby enabling rapid development of interesting virtual worlds.

The preferred embodiment of the present invention is adapted for use with the Microsoft Component Object Model ("COM"). In the Microsoft object-based technology ("OLE"), COM defines how OLE objects and their clients interact within processes or across process boundaries. However, one of skill in the art will readily recognize that the disclosure herein can also be applied to any other appropriate object-based technology and model. While the preferred embodiment of the Object System runs on Windows9X and Windows NT 4.0 and later versions, the invention can alternatively be implemented with any other suitable operating system.

The Object System specifies a set of COM interfaces which define the communication between the Object System and nodes. These interfaces are designed in such a way that nodes view data abstractly and do not call network functions directly. A central design feature of the preferred embodiment is that a node does not know at runtime whether it is executing as a consequence of an event originating on its local client or whether it is replicating behavior that originated on a remote client. A node programmer therefore does not explicitly have to address issues related to local or remote client behavior.

The Object System also defines interfaces for input/output components that are used to channel user input, for example using means including but not limited to mouse movements, keyboard input, and touch screens, into the world simulation and output, such as graphical/sound rendering out of the world simulation. These components will be referred to herein as *"ports."* By using ports, the Object System is isolated from its running environment. In addition, the use of ports greatly facilitates the addition of new types of input and output.

In general terms, ports are used to handle computer-oriented processes and elements such as graphics rendering, navigation, user login and pop-up menus, while nodes deal with processes and elements that are related to the world, such as cars, houses and cities. In addition,

nodes belong to the world and are *"shared."* The Object System is directed to keeping nodes that are running on different clients synchronized, so that each node appears to all participants to have a single identity and state. By contrast, ports belong to the client on which they run and their state is *"local"* (non-shared). Thus, two clients participating in the same world might not even have the same ports installed into their clients. Nodes and ports can be written in any programming language that supports Microsoft COM. As has been discussed previously, the invention can alternatively be implemented with any other suitable operating system and therefore, in alternative embodiments of the invention, nodes and ports are not required to be written in a programming language that supports Microsoft COM.

The displaying and rendering of spaces and dimensions is performed independently of the Object System. For example, a 3D world is displayed using a rendering port that is configured to interpret some of the world's nodes as 3D objects and to render them visually.

The Object System runs all nodes and ports on a single thread of execution. An Object System event mechanism is used to manage the execution of nodes and ports on the thread as well as to manage how nodes can control and respond to each other. In the preferred embodiment of the invention, this *"world simulation"* is driven by input ports, but is never affected by any output ports.

The Object System is connected to a remote World Server through a component known as the *"network abstraction layer"*. This network abstraction layer communicates with the World Server, which handles the lower-level activities of persistently storing and distributing world data. As a result, different network abstraction layers can be used. In most embodiments, the network abstraction layer communicates across a network with a remote (central) world server or network of servers. However, this is not a requirement. For purposes of this detailed description, the network abstraction layer will be referred to herein as "the server", regardless of whether the particular embodiment of the invention includes a remote server.

Figure 1 is a system diagram of the basic structure of a typical Object System client according to the present invention. The picture shows a client machine 100 connected to a remote server 102. There are 3 (connected) nodes in the world to which the client is connected: nodes 104, 106, and 108 There are 3 ports installed into the client: ports 114, 116 and 118. While the nodes and ports execute and communicate in the same process, the state of the nodes is shared while the ports are not.

In the client illustrated by Figure 1, port 114 knows about node 106 and controls it. Port 118 controls node 104 in a similar way. Port 116 and node 104 have bi-directional communications: port 116 controls node 104 while node 104 requests some services from port 116. Node 108 is not affected by any ports. In the present invention, node behavior executes on clients only.

The client application 120 has two user interface elements 122, 124 such as windows. Because the user interface elements are external to the Object System 130, they can only communicate only with ports. User interface ("UI") element 122 has bi-directional communications with port 114. UI element 122 might for example, be configured to interpret mouse clicks in the window and to send appropriate commands to the port. UI element 122 might also be configured to respond to changes in the node by updating indicators in the UI element. UI element 124 only responds to the actions of Port 118. For example, UI element 124 might be configured to display an image of the world rendered by Port 118.

The Object System 130 communicates with the network abstraction layer 140, which usually communicates through the network 142 with a remote server 102 running a World Server application 144. While the remote server is described in this example as being a single server, the remote server can alternatively comprise a hierarchy or a web of servers. The Object System is isolated from the server back-end configuration. The server passes data around but does not interpret it or execute any part of the world simulation.

Worlds, Nodes and Variables

A world is a container and a namespace for nodes. To participate in a simulation according to the present invention, a client must enter a world. The client passes a textual world name to the Object System, which passes it on to the world server component. The world server component must be able to resolve the name to a unique world and connect to this world.

Two clients must enter the same world to be able to interact with each other. A snapshot of all the values of a node at a particular time is called a *"node state."* The combined node states of all the nodes in a world constitute the *"world state."* The latest world state is persistently stored in the server's *"world database"* 150. Clients in the same world interact by changing the state of nodes in the world.

A node is the fundamental building block of a simulated world. New nodes can be created and destroyed in the world. Each node has a persistent *"node name"* that does not change during the node's lifetime. A node name is a bit string of fixed length that is generated by the server component when the node is created.

A node has one or more named *"variables"* that contain the world's data. Each variable is of a *"variable type,"* which dictates the interpretation (number, string etc.) and binary format of the variable's value (data). The variables of a node are ordered and known by their index, starting from 0. Each variable also has one or more textual names. In the preferred embodiment of the invention, there are several predefined type names from which to choose, corresponding to typical primitive data types in programming languages, such as integers, floating point numbers, and strings. Table 1 is a list of exemplary predefined type names and their corresponding data types.

TABLE 1

Type name	Data type
Boolean	Boolean (true / false)
(Unsigned) Integer	(Unsigned) integer, up to 32 bits
(Unsigned) Long Integer	(Unsigned) integer, up to 64 bits
Float	IEEE 32-bit floating point number
Double	IEEE 64-bit floating point number
String	Zero terminated 8-bit ASCII string
Wide String	Zero terminated 16-bit wide string
URL	Uniform Resource Locator
Binary Blob	Variable length buffer of arbitrary bytes
Reference	Nodename

The reference variable type contains the name of another node. The semantics of the reference variable type are similar to that of a pointer, allowing a node to contain a reference to another node. Because the references are directed, going from the node with the reference variable to the node named in the reference, the world comprises one or more directed graphs of nodes. References can contain the names of non-existent nodes and are therefore not guaranteed to be valid. A distinguished node name called the *"null nodename"* can be put as a value into a reference variable, to indicate that the reference is not referencing any node.

In the presently preferred embodiment, there are two basic kinds of variables: singleton variables and table variables. Singleton variables hold a single value while the table variables contain zero or more pairs of singleton values and allow entries to be retrieved or deleted by the first entry in the pair. The table variable thus functions essentially as a mapping or a look-up table, where the first entry in the pair is the key but the latter entry is the value. Alternatively, the table can contain no values but only keys. In this case, the table functions as a mathematical set, indicating only whether a key is in the variable or not. A table variable might, for example. map commodity names (strings) to prices (floats) or be a set containing references to nodes which it considers to be friends.

Variables can be tagged as "write only" variables referred to as "*event*" variables. Event variables are local, and therefore, when an event variable is set, that event is not replicated on remote clients. However, if an event variable in a node is set, the node might decide to change some shared state in itself or other nodes. An event variable is in many respects similar to a function call with one parameter.

Node Types

All nodes belong to a *"node type."* A node's type specifies what variables the node has, their order, their names, and their variable types. A node type approximately corresponds to the concept of object class or object type in programming languages. A node type can also be interpreted as a fixed interface that is used by nodes and ports to communicate. The definition of a type is fixed and should not change under normal operating conditions. A node's type is specified when the node is created. In the preferred embodiment, it is not possible to assign a different type to a node during its lifetime.

Each node type has a unique *"type identifier,"* which is a bit string of a fixed length. A type identifier encodes the location of a *"template implementation"* file for the type. The template implementation file contains the complete description of the type. As will be described in further detail, the template implementation is, in fact, an ordinary executable node implementation, and all node implementations are their own type template.

Template implementations are not a fixed part of either the world or the client and therefore can be stored in remote locations such as on a HyperText Transport Protocol ("http") server. This feature enables new types of nodes to be dynamically introduced to a world and/or client. When a type is first seen by a client, its template implementation is downloaded and registered with the client. Ports are also known by type identifiers, so new ports can be dynamically added in the same way.

Types can depend on other types. For example, if type A depends on type B, then the implementation file of type B will always be downloaded and registered before the implementation file of type A. If type B is a port, it will be automatically installed into the world, if needed. The Object System will fail to instantiate objects of type A unless type B has been successfully registered. Nodes can, for example, use this dependency mechanism to ensure that they will not be instantiated unless some specific port is installed into the world.

A node type usually communicates what a node is (a coordinate system, a physical object, a vehicle, etc.) and, at the same time, communicates the node's set of properties (a 4 by 4 matrix, weight, current speed, etc.). These attributes often imply what can be done with the node and how the node will behave. As an example, the scaling component of a coordinate system's matrix can be set, the weight of a physical object can change (when it burns, for example), and the speed of a vehicle can be increased or decreased.

A node often signifies some "concrete" reality such as mass or smell, but a node can also signify "conceptual" matters such as relationships and memories. In fact, what is concrete and what is conceptual is controlled by other nodes and the port rendering the world. For example, a graphical viewer might display images of the physical objects in a world, but a relationship viewer might display the world as a graph of connected nodes. When a first node meets a second node (by having a reference to it, for example), the first node can obtain the second node's type and behave according to it. If a paintbrush node meets a node which has a variable named "Color", it might set that variables value to Red, for example. A mouse node might destroy (eat) all cheese nodes it finds and add the value of their "Nutrition" variables to it's own "Energy" value.

The Object System supports a basic form of Object Oriented polymorphism. If node A has, for every variable in node B, a variable with the same name and type (the variables of B are a subset of the variables of A), then node A is said to belong to the type of B or to support the B interface. Node A can, for all practical purposes, be treated as if it were a type B node. However, node A does not necessarily support or duplicate the behavior of B, but only its type.

The Object System supports treating a node as if it were a node of a different type and for inquiring whether a node supports a specific interface. The Object System does not support the direct inheritance of node types or implementations. However, nodes can achieve similar effects by aggregating other node types. This is accomplished by a node's supporting another type's interface and "secretly" creating an node instance of that type to which events can be forwarded.

Basic Execution Model

Everything that occurs in a world is in response to a *"state update."* A state update is a change in the value of a node variable. State updates can trigger new state updates. As a result, a *"state update cascade"* can result from an initial state update.

All state updates and, therefore, all cascades originate from a port in one of the clients in the world. For example, a port might perform a state update in response to user input or because a software program running on the client's machine might give the port a command. A client that originates a cascade will be referred to herein as the *"original client"* with respect to that cascade. The other clients are the *"remote clients"* with respect to that same cascade.

It is the responsibility of the Object System to replicate each cascade from its local client on the remote clients. The state updates from a cascade are sent to the server. In response thereto, the server updates the world database with the changes. The server also forwards the state updates to the remote clients, which update their local copies of the world state with the changes.

The reason for a port executing an update is not significant for purposes of this discussion. Rather, it is significant that the port executes the update in response to some unique local event which is not duplicated on other clients. A user clicking a mouse button is an example of such a local event. This is distinguished from a global event such as a timer going off on all the clients in the simulation, causing all of them to behave like the "original" client and sending the update cascade to the server.

Node behavior is always associated with a state change. A node's code can basically only execute when the node's state is updated (when a variable in it gets a new value) and when state in some other node, which the node monitors, is updated. The program logic of the nodes (and ports) governs how an initial state update becomes a cascade of updates.

As an example, a port might change the value of a variable in a node A. When node A is notified of the change, it responds by changing a variable in another node B. Node B might in turn make changes to other nodes. Thus, a cascade forms a hierarchy of ports and nodes that call other ports and nodes. The Object System ensures that the call graph does not form a circle, which might lead to an infinite recursion.

A node monitors another nodes by referring to it through a reference variable. Thus, if node A has the name of node B in a reference variable, node A is notified of all state updates in

node B. This mechanism endows nodes with abstract "sensory" capabilities. Letting a node respond to another node might be used to simulate the node seeing or hearing the other node or responding to changes in it's position in space, for example.

Because ports do not have variables, they do not receive state changes. However, ports can monitor nodes and respond to state changes in them, as can nodes. Therefore that ports can both originate cascades and take part in other cascades. Because ports cannot have reference variables, the monitoring mechanism of ports is slightly different from that of nodes.

For purposes of explanation only, certain graphical notations will be used herein to describe state update cascades. Nodes will be represented by circles, while ports will be represented by boxes. Solid arrows leading right mean a state change on the node at the arrow's head, coming from the node or port at the arrow's tail. A dashed arrow leading left means the node or port at the arrow's head is responding to a state change in the node at the arrow's tail.

Figure 2 is an example of a state update cascade according to the present invention. In this example, port 202 responds to a mouse click 200 from a user by setting the variable *x* 204 in node 206. Node 206 responds by setting the variable *y* 208 in node 210. Node 212 is monitoring node 210 (has a reference to node 210) and responds to the change in node 210 by changing variable *z* 214 in itself.

Interactions between nodes are not described herein in terms of messages or events, but rather in terms of state updates. This is because in a persistently shared world that users can enter and leave at any point in time, an event that effects the world's state creates a state change. This state change must be permanently recorded in the world database or clients entering the simulation at a later time will get a different view of the world than the clients that saw the event occurring. Therefore, the Object System acknowledges only state and lets changes to the state play the role of local events.

The execution model according to the present invention implies that all nodes must follow a first rule that the behavior of a node must be completely governed by it's shared state. This is necessary if the simulation is to be replicated correctly. The Object System and server distribute state updates only, so the nodes must react the same way to the state updates on the remote clients as on the local client. For example, if a node A responds to its *x* variable being changed by changing variable *y* in node B and variable *z* in node D, node A must perform this exact sequence of actions on all clients participating in the world. Otherwise, the world

simulation will proceed differently from client to client resulting in the creation of multiple, divergent worlds.

Therefore, a node cannot "hide" local variables and let them influence its behavior, but must derive all its behavior from the node's shared state variables. In one embodiment of the invention a node can keep internal data for certain purposes, such as for optimization. However, in this embodiment, the node must be configured such that this internal data does not cause simulation divergence.

Because clients can enter into and out of simulations, and because a client entering a world must get the latest state of existing nodes from the server, the execution model also leads to a second rule that a node must be able correctly handle being initialized at any time with any valid state for the node.

These two rules should result in a node always behaving the same on all clients, even when clients enter and leave the world at different times and get different initial states for the node. The Object System helps with the implementation of the second rule by always setting variables in the same order, from first to last, when initializing nodes.

As has already been discussed, when a node is notified about a state change, the node does not know whether the state change is originating on the client on which the node executes or whether the update originated remotely but is being replicated locally. The node follows the same (world-domain) logic in both cases. By contrast, the Object System on the other hand does know whether the cascade is original or remote, and behaves accordingly. For example, when a state update occurs in a local cascade, the update is sent to the server. However, a state update that occurs in a remote cascade is carried out locally but is not sent to the server because a remote client has already sent the update to the server.

The Object System is configured with a set of rules to deal with more complex updating issues. In addition, the Object System is configured to use its knowledge about cascades to reduce network bandwidth requirements by not sending certain state updates to remote clients when the Object System knows that these state updates will be regenerated on the clients as a part of the cascade. These rules are used by the Object System to ensure that a state update cascade executes the same way on all clients and that the state update is sent to the server by exactly one client.

If all nodes and ports follow the rules these conditions are fulfilled, as long as no conflicts arise between cascades originating on different clients. In one embodiment of the present invention, nodes and ports are designed to withstand such conflicts and to minimize the likelihood of serious world divergence.

Many operations in the Object System are asynchronous. This is a result of the distributed nature of the simulation. When an asynchronous operation is completed, a callback function is called in the requester to notify the requester of the results. Setting a variable's value is one such operation.

The Object System has a synchronized global simulation time. In the presently preferred embodiment, the simulation time is synchronized with the world server's time to within one or two network latency periods to provide sufficient accuracy to synchronize actions such as the loosely synched playback of music or videos inside a world. In alternative embodiments, the simulation time can be synchronized to provide different levels of accuracy.

Choosing a Subset of the World

The Object System was designed to be able to handle very large worlds, with potentially hundreds or thousands of nodes and simultaneous users. However, because of the inadequate bandwidth available to many users, it is not practical for the Object System to attempt to distribute and update the state of all nodes in a large world. In addition, there is a limit to the amount and rate of data that can be processed by other world client subsystems, such as graphical renderers or physics simulators.

To overcome these limitations, the Object System uses the graph structure of worlds to select subsets of the worlds. The subset of nodes in a world that has been loaded by a particular client and who's behavior the client is executing, is called the client's *"active set."* The active set of a client can vary over time, for example as the user moves around in the world or enters new areas of the world.

The Object System provides a mechanism to help a client control the size of its active set. The client and/or the ports installed into it can map space-bound or other higher-level partition methods onto this mechanism, in a manner suitable to the semantics of the world. A client requests that a node be added to its active set by telling the Object System to mark it as a

seed node. The Object System forwards this request to the server, which responds by sending back the initial state of the node and any changes to it from that time on, thereby "activating" the node. A client can remove the seed status of a node, which might "deactivate" the node, dropping it from the active set. A node that is in a client's active set is active in that client, but inactive otherwise.

The Object System allows nodes to be configured to automatically come into the active set along with certain other nodes. Setting a node as a seed node can therefore result in many nodes entering the active set and removing seed status from a node might result in many nodes dropping out of the active set. To force node B to always be in the active set when node A is in the active set, node A must have a reference to node B. In addition, this reference variable must be marked as being "activating". Reference variables are non-activating by default, which means that the reference does not have any bearing on whether the referenced node is in the active set or not. References can be made activating or non-activating at runtime by both nodes and ports.

Activating references function transitively. If node A has an active reference to node B and node B has an active reference to node C, then all the three nodes will enter the active set when A does. Being "active" is a property of the reference variable and not the value stored in it. If a new node name is put into an activating reference variable, the node bearing that name will enter the active set (if it exists and is not already active) and the node previously referenced in the variable (if any) might become inactive.

The active set calculation is performed on the server. A rule that generally describes which nodes should be active on a client is that a node is active if and only if it is a seed node or if it is referenced by an active reference in an active node.

Whenever a state update cascade is sent to the server, the server determines whether the update affects the current active set of each client and, if so, notifies the client as to any changes, such as nodes entering or leaving the active set. When a node becomes inactive, the client stops executing it and the server stops sending the client state updates for it.

Figures 3a-c illustrate changes to an active set according to the present invention. In Figures 3a-c, nodes with a heavy border are seed nodes, thick arrows are activating references, and thin arrows are non-activating references. Active nodes are shaded.

While there are 4 seed nodes 300, 310, 320, 330 shown in Figure 3a, a client can have one or more seed nodes. For example, a typical client may have only a single seed node representing the user's avatar. An avatar is the representation of a user in the shared world, often a humanoid figure. The avatar can have motions such as "walk" and "dance". The active set shown in Figure 3a is disjoint- the two rightmost active nodes 330, 332 do not have any connection to the left part of the active set. The Figure also includes a completely unconnected node 302.

Figure 3b illustrates the changes that are made to the active set shown in Figure 3a if seed node 300 loses its active status. In Figure 3a, seed node 300 has an activating reference to node 304, which itself has an activating reference to node 306. Node 306, in turn, has an activating reference to node 308, which has an activating reference to seed node 320. As can be seen in Figure 3b, node 304 is no longer in the active set because seed node 300, which had previously actively referenced node 304, has been deactivated. Similarly, nodes 306 and 308 have been deleted from the active set. However, because node 320 is a seed node, it is not deactivated when node 308, which had previously actively referenced node 320, is deactivated.

Figure 3c illustrates the changes that are made to the active set shown in Figure 3a if the active link between nodes 304 and 306 becomes non-activating. Nodes 306 and 308 have been removed from the active set in response to the deactivation of this link.

The active set rule implies that a client must have at least one seed node if it is going to have a non-empty active set. If a client wants to enter a world, it must know the name of at least one node in the world or have some way of obtaining that name. The numbers of activating reference variables in a world can be adjusted to optimize the amount of active nodes. In addition, changing the active condition of reference variables can be incorporated as a part of port or node logic. For example, a door can be given have references to the rooms on each side of it and these references made active if and only if the door is open.

Interfaces and Proxies

A node is a COM object that implements the IObsNodeImpl interface. The Object System calls functions in this interface to initialize the node, to get information about its type, to set and get the values of its variables, and to notify it about certain events in the world, such as

state changes in other nodes which the node references. Ports, in a similar way, must implement the IObsPortImpl interface, which has most of the same functions as IObsNodeImpl.

Nodes in a world are managed by a *"world object."* To enable node and port implementations to talk back to the world in which they live, the world object provides IObsInsideWorld objects. These objects are available to entities that execute inside the world, namely, nodes and ports.

The world object is instantiated by a client application. When a client application wants to connect to a world, it creates the Object System *"world factory,"* which is a single-instance COM object. Whenever such single-instance COM object is created, a pointer to the same instance of the object is returned. The world factory has the IObsSystem interface. This interface includes functions to open existing worlds and to create new ones. Both these functions return an IObsWorld interface to the new world object, if successful. The main visible Object System entity is the world object.

The IObsWorld interface is the only means of communication between the world object and external applications. It is a very limited interface, and the state of the world cannot be directly changed through it. The main operation an application can perform through the IObsWorld interface is to add a new port to the world object. When an application adds a port to a world object, it receives in return a pointer to an arbitrary (port-defined) interface on the port. The application and the port can then communicate through that interface.

Nodes and ports inside a world can find ports in their world object and query them for arbitrary interfaces. On the other hand, nodes and ports cannot get arbitrary interfaces to node implementations. Only the Object System can talk directly to node implementations. When a node or a port has obtained a node name and wants to query or modify that node, it asks the Object System to give it a *"node proxy"* for the node.

Node proxies are COM objects with the IObsNode interface. As does the IObsNodeImpl interface, the IObsNode interface has functions to get information about the nodes type and to set and get the values of its variables. When a function in a proxy is called, it forwards the call to the corresponding function in the real node implementation. Similarly, the world object has proxies known as IObsInsideWorld objects. These objects will be referred to herein as *"inside-world proxies."*

Because all state updates go through a node proxy that the Object System implements, the Object System can see all state updates as they happen. Data distribution and persistence issues are transparent to the nodes getting changed and the nodes and ports doing the changes. This is because the nodes and ports just update variables and the Object System handles the other related issues. The Object System also performs extensive error checking on all function calls. This increases the robustness of world clients and makes development easier because node developers can assume that the node only gets valid function calls.

Both node and inside-world proxies are marked with their owner, which is the port or node which originally asked for them. When nodes and ports call functions in proxies, for example, to create or destroy nodes or change variables, the proxy and thus the world object, knows who is calling. This enables the Object System to return notifications about operation completion to the right caller, and to handle cascade replications correctly for a wide variety of cases.

Figure 4 is a diagram showing an example of object and interfaces in a client according to the present invention. In the Figure, objects are shown as rounded rectangles, and interfaces as circles on sticks extending from objects. Arrows show pointers to interfaces from the object holding the pointer to the interface on the referenced object. The objects that are provided by the Object System have a thin border, while application and/or world specific objects have a thick border and bold font.

The application 400 has a pointer to the IObsWorld interface 450 of the world object 402. For purposes of this diagram, it is not important whether the application created this world object or instantiated it from an existing world on some remote server. The application has added two ports to the world object, the Input Port 406 (handles mouse & keyboard) and the Output Port 404 (handles sound and graphics). The application also holds pointers to interfaces 454, 456 on the ports that are particular to the parts and the application. There are currently two nodes in the active set – Node 1 (414) and Node 2 (416).

The world object has pointers to the IObsPortImpl interfaces 458, 460 on the ports, through which it lets ports know about various events in the world. The ports have inside-world proxies 408, 410 to the world object, through which the ports can perform operations on the world. The ports have pointers to IObsInsideWorld interfaces 480, 482. The input port has a proxy 412 for Node 2, through which it can change the state of Node 2, for example, by moving

Node 2 around when the user moves the mouse. The input port has a pointer the IObsNode interface 484. The output port in this client does not have proxies to any nodes at the instant represented by the Figure.

The world object has pointers to the IObsNodeImpl interfaces 470, 472 of the nodes. The nodes have pointers to the IObsInsideWorld interfaces 490, 492 to inside-world proxies 418, 422 for the world object. Both the nodes also have pointers to IObsNode interfaces 494, 496 to node proxies 420, 426 to themselves. This is because nodes can only change state, including their own state, through IObsNode in a node proxy. If nodes were to change their state internally without going through proxies, the Object System would not know about the changes and would therefore not be able to distribute or replicate them.

Node 2 has a pointer to the IObsNode interface 498 to proxy 424 to Node 1. Such a proxy can be used, for example, to permit Node 2 to monitor Node 1 or to change its state under certain circumstances in response to Node 1. Node 1 has a pointer to an arbitrary interface 430 on the output port. Thus, for example if Node 1 were "visible", it would be able to send the port information as to how it should be displayed. All of the Object System-provided objects 408, 410, 412, 418, 420, 422, 424, 426 have private links (not shown) to and from the world object 402.

If a node or port has a pointer to a proxy interface, it must not hand that interface pointer to any other node or port. Therefore, for example, if a proxy is originally handed to Node A or Port A, the Object System will always assume all calls on it originate on Node A, and send the resulting callbacks to Node A alone.

The Object System's simple polymorphism is based on proxies. A requested proxy to a node can be of some other node type than the node. If the node can, in fact. be cast to that other type, the proxy will be created. Otherwise, the request will fail. For example, a world can include a node type X with integer variables e, f, g and h, and node type Y with integer variables *e* and *g*. Because the variables of node type Y are a subset of node type X, Y proxies can be created to X nodes. Such Y proxy allows the actual X node to be treated as if it were a Y node. This feature is advantageous, because variables are known by an integer index and therefore this process can be automated, rather than performed as a manual mapping.

The Basic Simulation Cycle

The Object System simulation is single threaded. A world object and all the nodes and ports in it run on a single thread of execution. Worker threads can be created for tasks such as downloading data as long as these threads do not directly affect the simulation. The world object drives a world's simulation and, therefore, nodes and ports do not usually communicate unless first contacted by the world object. The world object notifies the nodes and ports about certain events in the world and the ports and nodes can respond if they want to by performing operations on the nodes in the world.

In the presently preferred embodiment, there are six types of notifications, known as *"callbacks"* that are applicable to nodes and/or ports:

SetValue: to let a node know that it's state has been changed.

ValueChanged: to let a node or port know that the state of a node to which the port or node refers has changed.

NodeDestroyed: to let a node or port know that a node to which the port or node refers was destroyed.

NodeActiveChanged: to let a port know about a node to which the port refers entering or leaving the active set.

NodeSpied: to let a port know about certain nodes entering the active set.

Tick: to let a port know that a certain amount of time has passed.

The Tick callback is significant because most changes in the world result from a Tick on some client. The Tick is the Object System's simulation's replacement for the passing of time.

Nodes receive notifications for nodes to which they have a reference, regardless or not of whether they have proxies to them. However, Ports only receive notifications about nodes to which they have a proxy. Ports can monitor nodes as the nodes enter the active set by asking for NodeSpied callbacks. When the NodeSpied callback is registered, the port can specify whether it wants to be notified about all nodes entering the active set or only nodes that can be cast to a certain type. A sound port, for example, might wish to only be notified about nodes which of the "sound emitter" node type. When the port receives the NodeSpied callback, it must obtain a proxy to the node if it wants to keep on getting notifications about it.

Managing and Connecting to Worlds

To create a new world, the name of the server where the world is to be created and the name of the world must be specified. The name of the world is an arbitrary string. If the server is not found or a world with the same name already exists on the server, the operation fails. The Object System does not interpret the server name and makes no assumptions about it, it just passes it on to the server component. The server name can be a DNS name like "www.servername.com", an IP number like "1.23.45.67" or even the name of a local directory like "C:\Worlds" for a purely local server component. This operation can be blocked for a period of time while the server component attempts to connect over the network to a remote server. If the operation succeeds, the caller receives an interface pointer to the IObsWorld interface of the new world object.

The same parameters are provided to open an existing world as when creating a world. If the server is not found or the world name is not found on the server, the operation fails. If the operation succeeds, the caller gets an interface pointer to the IObsWorld interface of the world object for the existing world.

In the presently preferred embodiment, both the create world and open world operations accept arrays of storage identifier definitions that map storage identifiers (which are parts of node/port type identifiers) to file locations. The world object can only download and use those nodes and ports whose storage identifier is defined in the array passed in the operation. At least one such storage identifier to location mapping must be defined, in which case, all the nodes and ports must have that same storage identifier.

A close world operation can be performed to disconnect the client from the server and close the world. All ports and active nodes will be released at this time.

External (Local) Operations on World Objects

In the preferred embodiment of the present invention, the following operations can be performed by external applications that hold a pointer to the IObsWorld interface:

Add a New Port: To add a new port, the caller passes in the type identifier of the port that is to be created and inserted into the world object. The caller also specifies the interface it wants to

get from the newly added port. The operation fails and no port is added if the port type can't be resolved or if the port doesn't support the requested interface. The port is being added to the local world object and adding a port is a purely local operation. Other clients on other machines and in the same world might have different ports installed into their respective world objects.

Get an Existing Port: Outside applications can search for ports by type identifier or by type identifier and interface. If a type identifier is specified, only ports of that type are included in the search. Otherwise, any port supporting the specified interface is considered. If more than one port has the specified type identifier, the most recently installed port is returned.

Get Sundry Information: This operation permits selected inquiries concerning the world object. For example, application can use this operation to get the current simulation time, the world's name and server, as well as certain performance statistics.

Operations on Worlds

In the preferred embodiment of the present invention, the following operations are available to nodes and/or ports only through the IObsInsideWorld interface. These methods operate at the world level, for example creating nodes. Only the first two of the operations cause actual changes to the shared world state.

Create a New Node: When a port creates a new node, it gets a special callback. However, a node does not get a special callback when the node has been created. Rather, the node must pass in the index of a reference variable in themselves, where the name of the new node will be put. In both cases, the type identifier of the new node is provided. The node will have default values for all its variables when its creation is complete. All tables will be empty and all reference variables will be non-activating.

Destroy a Node: The name of the node to destroy is passed in as a parameter. If the node had active references to other nodes, those nodes which are only in the active set because of the node (not being seed nodes and not having other active references to them from other active nodes) will go out of the active set just after the node is destroyed, and so on recursively. If the node

being destroyed is not in the active set, it will be activated just before its destruction. The following objects will be notified when the destruction completes:

All nodes that have a reference to the node (in a reference variable)
All ports which have a proxy to the node
The node being destroyed.

Remove a Port: While ports typically reside in the world object once installed, they can request to be removed from the world object. Removing a port is a local operation and therefore only affects the local world object.

Add a Seed to a Node: The main way to bring a node into the active set is to mark it as a seed node. The name of the node is passed in as a parameter. This operation will be referred to herein as "adding a seed to a node" because node seeds are actually reference counted. Each time someone sets a node as a seed the node's seed count is incremented and each time someone removes the seed status from a node the counter is decremented. It is therefore appropriate to talk about seeds being "added" or "removed" from a node, which simply means that the node's seed counter is being incremented or decremented.

If the node to which the seed is added is already in the active set, its seed counter is incremented and nothing remarkable occurs. If the node is not in the active set, a request is sent to the server to set the node as a seed node. If no node with the specified name exists or if removing a seed fails, the caller will be notified through a callback.

If the node exists, it will enter the active set when the server notifies the Object System that the operation has completed. A new instance of the node will be created and initialized with its current state, which the server sends along with the notification about the node entering the active set. The following objects will be notified about the change:

All nodes which have a reference to the node in a reference variable
All ports which have a proxy to the node
All ports which are spying on a type which the node supports.

If the node entering the active set has active references to other nodes, these will enter the active set just before the node itself, and so on recursively. The dependent nodes enter the active set in approximately bottom-to-top order. If the node type depends on other types, these types will be installed before the node gets initialized. Adding a seed is a local operation and, therefore, remote clients are not aware of or affected by the local client's seeds.

Remove a Seed from a Node: Removing a seed decrements a node's seed counter. If the counter hits zero, a request is sent to the server to remove seed status from the node. The name of the node is passed in as a parameter. If the node doesn't exist, the caller is notified through a callback. If the node does exist, it will be removed from the active set when the server notifies the Object System that the operation has completed. The following objects will be notified about the change:

All nodes that have a reference to the node in a reference variable

All ports that have a proxy to the node.

If the node leaving the active set had active references to other nodes, those nodes which are only in the active set because of the node (not being seed nodes and not having other active references to them from other active nodes) will go out of the active set just after it, and so on recursively. The dependent nodes leave the active set in approximately top-to-bottom order. Removing a seed is a local operation and, therefore, remote clients are not aware of or affected by the local client's seeds.

Spy on Nodes (ports only): Ports can request to be informed whenever a new node enters the active set. The port can limit the notifications to nodes which can be cast to a certain node type. When a port gets a callback informing it that a node is becoming active, it must get a proxy to the node if it wants to keep on monitoring it, otherwise it will receive no further notifications about the node. When a port requests to be informed about a node, it will get callbacks for all matching nodes which are already in the active set, before returning from the call. Spying on nodes is a local operation and therefore it only affects the local client.

Get a Node Proxy: If a node or port wants to get or set the variables of a selected node, it must first get a node proxy to the node. It passes in as a parameter the name of the selected node. It can either get a proxy to the native or full type of the selected node or get a cast proxy to a subset of the selected node's type. If the cast is not possible or the selected node is not in the active set, the operation will fail.

Getting a cast proxy can be used as a test to see if a node supports a specific type. If and only if you succeed in getting a type X proxy to some node, you will know that the node supports type X. Once you have a type X proxy to the node you can use the proxy as if the node really were a type X node. The proxy will map all variable indices to the indices of the node's real type. Getting a proxy is a local operation that only affects the local client.

Get an Existing Port: Nodes and ports inside a world can search for ports in a similar manner as applications outside the world. Getting a port is a local operation that only affects the local client.

Register for Ticks (ports only): Ports can request to be periodically notified, or to get timer "ticks". The desired interval is passed in as a parameter. In one embodiment of the invention, the interval is rounded up to the world simulation's maximum time granularity. A "context" integer can also be specified, which will be passed in with the Tick callback to help ports maintain many simultaneous "channels" of ticks. To unregister a tick channel, a negative interval is specified, with the context of the channel. Registering for ticks is a local operation and only affects the local client.

Get Sundry Information: A port or node can inquire about the state of another node. A port or node can also get the current simulation time. In all operations that accept a type identifier as a parameter, if the type identifier cannot be resolved, the operation will fail. The presently preferred embodiment of the Object System does not download nodes so the type's implementation file must exist locally. However, in alternative embodiments, the type will be automatically downloaded if it is not found locally.

Operations on Nodes

The following operations are available to nodes and/or ports only. The methods operate on the node level, changing the attributes of individual nodes.

Get information about a proxy: From a node proxy, one can get its node name, the proxy's type, the number of variables in the proxy type, the names and types of these variables and help text explaining the type. A function can also be provided to search for a variable name and get its index. With the exception of the node name, the information applies to the proxy type, and not to the node's real type (except if the proxy is not a cast proxy but a proxy to the real/full type of the node).

Get a variable's value: When setting or getting values from a table variable, a key must always be specified, but when setting or getting values from singleton variables a zero pointer is passed in for the key. Keys and values are passed back and forth as pointers to "raw" byte buffers. When getting a value or a key, it is the responsibility of the caller to provide a large enough buffer for it. A simple way to do this is to pass in a buffer with at least the number of bytes equal to the largest allowable data value in a node variable.

The caller passes in a pointer to an unsigned integer. If the function returns successfully, the integer will contain the number of bytes that were copied to the value buffer. If the variable is a table variable and the key which was passed in was not found, a false indicator is returned. If the operation was successful and the key did exist, a true indicator is returned. If the table variable is in fact a set (the value type is specified as "none"), the only information that is returned is the return value, indicating whether the key is in the set or not.

The presently preferred embodiment includes two additional functions that only apply to table variables. The first function returns the number of entries in a table variable, while the second function facilitates iteration of the table's keys (which can in turn be used to retrieve the value for each key).

Change a variable's value: If the variable is a table variable, a valid key must be passed in. If the key was not previously in the table, it will be inserted into it along with the value, if the variable is not a set, otherwise the old value of the key will be overwritten with the new value.

In the preferred embodiment, an additional function that applies to table variables only is available to remove a key (and it's value, if the variable is not a set) from a table.

Inquire Whether a Reference Variable Is Activating: This operation permits a caller to check whether a reference variable is activating. Activating reference variables have been discussed previously in detail with respect to Choosing a Subset of the World.

Change Whether a Reference Variable Is Activating: This operation only applies to reference variables. Making active a reference variable that was previously inactive might trigger new nodes to enter the active set. Conversely, making inactive a reference variable that was previously active might trigger nodes to drop out of the active set. If a table variable has node references as keys or values only, the operation applies to the keys/values respectively. If a table variable has node references as both keys and values, the operation is taken to refer to the values only, not the keys.

Map a Variable Index: In changed value notifications, the variable index passed in is always in terms of the full type of the node that changed, and not in terms of any subtype proxies. For example, if a port is spying on all nodes which can be cast to type X, and the node gets an X proxy to a node of type Y (which is a superset of type X), the node will get notifications in terms of the full type Y, even if it is spying on the X type. The port therefore has to map Y variable indices to X indexes, which it understands.

This function is available to map a variable index from the real type of the proxy's underlying node to the type of the proxy. In the above example, the port would get a notification with a variable index in Y (which the port doesn't understand). However, after mapping it with a X proxy, the variable index is now in terms of X (which the port does understand).

The Lifetime of a Node Instance

The lifetime of a node and the lifetime of its executable node instance on a particular client do not usually coincide, although the latter is always a subset of the former. Node instances are temporary incarnations of the persistent node.

A node can enter and leave the active set on a client many times during a run. Each time it enters the active set, a new node instance is created and initialized with the node's name and state. Each time it leaves the active set, the instance is destroyed. Only the client's node instance is being destroyed, not the actual node.

A node entering or leaving an active set is not a change in the shared state of the world and does not constitute an "event" like a variable getting a new value. When a node instance is initialized as a consequence of becoming active, it must immediately start behaving according to its state, just as if it had always been actively running. For example, the node instance cannot change the world's state just because it is entering the active set on a particular client. This also applies when a node instance is released. However, because the actual creation or destruction of a node is a real event in the shared world, the node can perform operations at these times.

Figure 5 is a flow chart of the initialization phase of a new node instance according to the present invention. After the node instance has been created 510, the Object System asks the node instance about its type 520. In the presently preferred embodiment of the invention, the Object System caches the type information for later reuse so step 520 is only performed in the first instance of a particular type. The node instance next gets state information 530 for all variables that have changed from their default values. To conserve bandwidth, the world server only stores variables that have actually been altered. All other variables are assumed to have default values. The node instance is initialized 540, receiving the node's name and a proxy for it. After this, the node instance is ready to respond to world events and execute normally. When the node leaves the active set, it is released without further action.

Examples

Following are examples of different client and world setups, ports and nodes according to the present invention.

Example 1 - A Standalone Simple Client

Figure 6 is a diagram of an exemplary simple Object System client 600 according to the present invention. The client application 620 has a single window 610 that is maintained by the application. The application gets mouse, keyboard and menu input from the window, interprets it and passes it to the input port 630, which passes it on to the relevant node(s). The application can pass either "raw" user interface ("UI") events to the port and let the port interpret them or it can interpret the events and then pass them to the port, or both. In Example 1, the input port is specifically created for this client and is not reusable elsewhere. The output port 640 is also custom made for this client. The output port receives notifications from the client application about certain events in the world and renders them into the window. The world object 650 contains the active set 670 comprising nodes 672, 674, 676, 678 and communicates with a server component 660.

Example 2 - A Web Browser Embedded Client

Figure 7 is a diagram of an exemplary web-browser embedded client 700 according to the present invention. In this embodiment, there is no standalone application. Rather, several Web Browser plug-ins 790 (ActiveX or Netscape Plug-ins, for example) interact with a world object 750 through ports. The behavior and functionality of the clients is entirely determined by the interplay of the viewer controls, the ports in the client and the nodes in the world. Also, the controls and ports in this client are generic and reusable. They are designed to work with certain node types but can be used with different types of clients.

The layout of the controls and the connections are encoded in a HTML page. In alternative embodiments of the invention, any other suitable mark-up language can be used, including but not limited to XML and DHTML. An Object System control, the Page Manager 710 is used to initialize and connect the components on the HTML page. In one embodiment of

the invention, the Page Manager is also operable to download missing components from the HTML page.

The client has two viewer controls 720, 730, that are used to display two different views into the world, for example a 2D or 3D view. Each viewer control has a dedicated installed rendering port 740, 746 because different parts of the world are being rendered in each control. However, the viewer controls share an input port 742 and sound port 744. Only the viewer control currently holding the input focus communicates with the input and sound ports.

The rendering ports spy on nodes supporting a certain type. They get notifications when the state of these entities changes such as when an object moves or disappears, and render a new image into the control to show the new state of the world. The sound port, in a similar manner, monitors sound nodes and plays their sound clips at appropriate volumes.

There is also a purely internal "Geometry Port" 780, that provides services to nodes and ports only and not to the application-level controls. Services provided by the geometry port include but are not limited to assisting nodes in working with geometry hierarchies, and detecting geometry intersections (collisions). The world object 750 contains the active set 770 comprising nodes 772, 774, 775, 776, 778, 779 and communicates with a server component 760.

Example 3 - A Shared Whiteboard

A shared whiteboard is a drawing program that many persons on networked computers can use to edit the same drawing simultaneously, giving the illusion of a single shared picture. The shared-whiteboard example can be implemented with the simple client setup described previously with respect to Example 1.

Figure 8 is a diagram of the graphic node and port types in an exemplary shared whiteboard according to the present invention. The drawing primitives of the simple client shown in the example are circles, boxes, free-draws and text boxes. A free-draw is an ordered list of vertices that are connected with line segments.

The nodes 810, 820, 830, 840 are mostly data containers and therefore they do not exhibit much behavior. Most of the behavior of the whiteboard application is implemented in the whiteboard drawing port 850 and the whiteboard editing port 860. In the presently preferred embodiment of the invention, variable types are stored as binary blobs. For example, each blob

in the "vertices" table 830 can contain two 4-byte floating point numbers. Alternative embodiments of the invention support composite variable types, such as RGB triplets and *n*D vertices.

The drawing port monitors all circles, boxes, free-draws and text-boxes in the world (the whiteboard) and renders them onto a drawing area such as a Microsoft Windows™ Device Context or X-Windows Canvas. The whiteboard application has a simple viewer window/control (not shown) in which the shared drawing is displayed. When the window needs to be redrawn it notifies the drawing port, which draws it on the next Object System tick. The drawing port also redraws the window when a graphic node changes, for example because a remote user edited it. The drawing port can be configured to optimize the refreshing of the window by only redrawing the portion of the picture which has changed.

The editing port receives mouse events in viewport coordinates and translates them into editing commands on the whiteboard's graphic nodes. Thus, when the editing port gets a "left mouse button down" event, along with mouse coordinates, the editing port determines, for example, whether the mouse hit a vertex in a free draw, was inside a circle, or was on the border of a box. If a mouse click hit is followed by mouse-move events before the button is released again, the editing port can be used to perform actions including but not limited to adding, deleting, or changing vertices, moving or resizing a destination node. The Object System and server replicate the state changes on the remote clients so that the same drawing is displayed to all users.

Example 3 - A Shared Virtual 3D World

Example 3 is a simplified description of a feature-rich, shared 3D virtual world. An efficient and well-structured virtual world requires several basic nodes and ports. These basic objects are the building blocks of virtual worlds.

A new virtual world will typically use a set of basic generic ports and nodes, in addition to special custom-made nodes specific to the world. For example, a tank battlefield simulator might require for some tank and helicopter nodes, and a world with a virtual economy might require bank and monetary nodes. The creator of the world can decide whether to make new custom nodes or to adapt reusable nodes.

Figure 9 is a diagram of the ports and nodes of an exemplary simplified 3D client according to the present invention. The Object node type 900 is an "abstract" node because it has no behavior and is probably never instantiated. It simply defines a type into which other nodes and ports can cast. The Object node type defines the properties of a volumetric object in 3D space - it's size, position and orientation.

The Entity node type 930 has all the variables of Object node has and can therefore be cast into an Object node. In addition to the Object variables, it has a URL to a file containing the polygon geometry for the node. It also has "parent" 932 and "child" 934 fields to enable the creation of hierarchies of entities, such as a crate containing other entities. The logic in Entity ensures that the hierarchy is maintained. When a child is removed from the children set, for example, the parent field in the formed child node is also changed.

The Animation Entity node type 910 is an Entity node that can show animated geometry. It has all the variables of an Entity node, plus a set of animations that can be triggered by setting the "current animation" variable 912. An animation node can be used to create avatars for users, for example.

The Point Of View (POV) node 940 represents the "eye" or "camera" of a client peeking into the world. The POV node adds seeds to nodes and thus is significant in determining the client's active set. It has a "parent" node reference field 942 and it follows the node which is set as its parent around, pointing at it. The POV node only follows Object nodes. In alternative embodiments of the invention the POV node can follow the parent in different ways such as by remaining stationary at the parent node, or by flying around the parent node. For example, the POV node can be configured to behave such that it positions itself two units "behind" (along the Z axis) its parent and never rotates itself.

The Light node 920 represents a light source in the world. It can be turned on and off by setting the "on" variable 922 to true or false, respectively. The Light node is a simple node that only stores data and has no behavior.

The Door node is a specific-purpose node that represents doors in the world. It is an Entity node with the additional variables "closed" and "mouseclick". The former controls whether the door is open or closed. The latter is an event variable that is used to allow the door node to receive mouse events from an input port. By declaring the variable, the Door node

advertises to input ports that it is willing to receive mouse events. When it gets a mouse click, it switches from being open to closed or from closed to being open.

The Login port 960 handles the initial entry of a new client into a world. It decides the starting node for the client and sets that node as a seed node. It then creates an Animated Entity node to be the client's avatar and inserts it as a child under the starting node. It also removes the client's avatar when the world is shut down.

The viewer port 970 renders a 2D image of the world. It is initialized with the name of the POV node from which to render the image. It spies on all Entity and Light nodes and its assigned POV node. In the presently preferred embodiment, the viewer port downloads the rendering geometry for the nodes from a URL specified in the node's URL variable and then displays it. The viewer port also sends raw mouse events to the current Navigation port 980. The Navigation port interprets mouse events and moves some Entity around, usually the user's avatar. The Navigation port uses the services of the Bounding Volume port 990 to detect and resolve collisions. It is possible to switch navigation ports on the fly, for example to switch from a "walking" navigation to "driving" navigation.

The Bounding Volume Port keeps track of all bounding boxes in all Object nodes and maintains data structures to be able to efficiently answer intersection queries. It is a purely internal service, and outside applications can not talk to the Bounding Volume port. The Bounding Volume port does not use the actual 3D geometry of entities, only the bounding boxes. This is because the behavior of nodes is not affected by platform- or library dependent artifacts. The client can, in fact, run without an installed 3D viewer and rendering library. The nodes themselves do not interface with the renderer so they run regardless of whether they are being rendered or not. The geometry specified in the URL variable of Entity nodes is only used by the 3D viewer ports. If no 3D viewers are running, the geometry isn't downloaded, for example in a text MUD-like client viewing the world, for example. The geometry is purely decorative, essential to view the world but having absolutely no bearing on the behavior of the world simulation. This also applies to other types of files, including but not limited to sound files and video files.

In Example 3, the world's "true" 3D geometry essentially comprises oriented boxes. More advanced representation of the world's geometry can be used, however, the real structure

of the world, which drives the simulation, is not coupled to the structures used to visualize the world on a particular client.

Although the present invention has been described with reference to specific exemplary embodiments, it will be evident that various modifications and changes may be made to these embodiments without departing from the broader spirit and scope of the invention as set forth in the claims. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.

CLAIMS

What is claimed is:

1. A method for distributing a real-time simulation, comprising the steps of:

running a server software application on at least one server;

hosting at least one world on the server, the world comprising at least one node having at least one variable;

connecting at least one client through an electronic network to the hosted world on the server;

running at least one client software application on each connected client, the client software application operable to receive a current value of the variable, the client software application further operable to create and remove nodes in the world;

using the server to record in a world database changes to values of variables of nodes in the world;

subscribing each client to a subset of the world's nodes;

wherein, for any change in value of a variable in a node of a client's subset, the server sends notification in real-time to the client to update the variable.

2. The method of claim 1 further comprising the step of permitting the client to modify the subscribed subset of the world's nodes.

3. The method of claim 1 further comprising the step of providing the node with at least one reference to a reference node.

4. The method of claim 3 further comprising the step of updating the reference node in response to a change of value of a variable of the node.

5. The method of claim 1, wherein the step of running at least one client software application further comprises the steps of:

handling general chores of replicating worlds across clients using an Object System component; and

providing world-specific behavior using at least one component compatible with the Object System.

6. The method of claim 5, further comprising the step of connecting the Object System to a world server component for handling lower-level activities of persistently storing and distributing world data.

7. The method of claim 1, further comprising the steps of:
associating behavior with the node; and
executing the node behavior only on the client.

8. The method of claim 7, further comprising the steps of:

ensuring, using the Object System, that node behavior initiated on one client is replicated on other clients that have that node in their subscribed subsets; and

synchronizing nodes that are running on different clients using the Object System.

9. The method of claim 5, further comprising the steps of:

specifying, using the Object System, at least one interface to define communication between the Object System and the node; and

defining, using the Object System, at least one port for channeling input from the world and output to a user of the client.

10. A system for transmitting data over an electronic network comprising:

at least one server computer;

at least one world simulation database accessible to the server computer, the world simulation database comprising at least one node having at least one variable;

at least one server software application accessible to the server computer and to the world simulation database for hosting a world simulation and for recording in the world simulation database all changes to values of variables of nodes in the world;

at least one client computer adapted for connection over the electronic network with the world simulation hosted by the server application;

at least one client software application on the connected client, the client software application operable to receive a current value of the variable, the client software application further operable to create and remove nodes in the world;

at least one input/output component connected to the client for communicating with the world simulation database;

an Object System component accessible to the client for subscribing the client to a subset of the world's nodes and for handling general chores of replicating worlds across any client connected to the; and

at least one component compatible with the Object System for providing world-specific behavior;

a world server component accessible to the client for handling lower-level activities of persistently storing and distributing world simulation data;

wherein, for any change in value of a variable in a node of a client's subset, the server sends notification in real-time to the client to update the variable.

11. The system of claim 10, wherein the node comprises at least one reference to a reference node, wherein the reference node is updated in response to a change of value of a variable of the node.

12. The system of claim 11, further comprising:

at least one interface defining communication between the Object System and the node; and

at least one port defining communication between the input/output component and the Object System.

13. A client for use in a distributed real-time world simulation, comprising:

a world object comprising an active set of at least one node;

a client application;

a window maintained by the client application for receiving input and transmitting the input to the client application;

at least one input port for receiving the input and transmitting the input to the node;

at least one output port for receiving at least one notification from the client application and for rendering the notification into the window; and

a server component in communication with the world object.

14. A web browser embedded client for use in a distributed real-time world simulation, comprising:

a world object comprising an active set of at least one node;

at least one web browser plug-in module for interacting with the world object;

at least one viewer control for displaying a view into the world;

a dedicated installed rendering port for the viewer control, the rendering port operable to spy on a selected node of the active set;

at least one input port for receiving input from the viewer control and transmitting the input to the node;

a geometry port for providing services to the at least one node and the ports; and

a server component in communication with the world object;

wherein the layout of the controls and connections of the client are encoded in an HTML page.

15. The client of claim 14, further comprising a page manager component for initializing and connecting components on the HTML page and for downloading missing components from the HTML page.

16. The client of claim 14, further comprising a sound port for monitoring and playing sound data from at least one sound node in the active set.

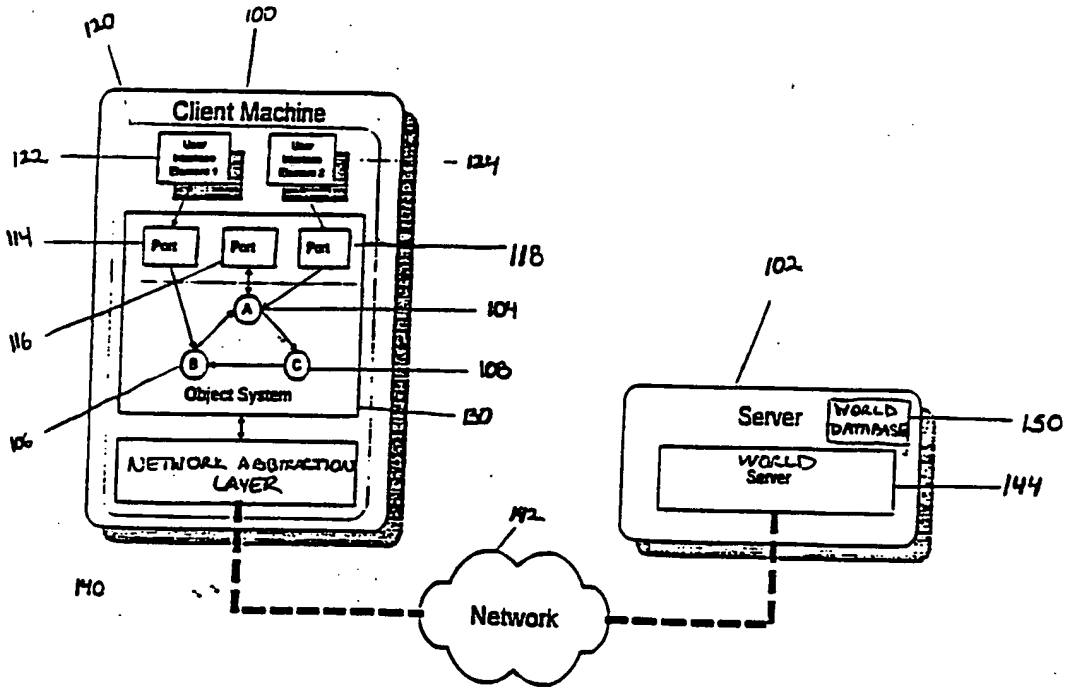

120
100
Client Machine
122
124
114
Port
Port
Port
118
116
A
104
B
C
106
108
Object System
150
NETWORK ABSTRACTION LAYER
102
Server
WORLD DATABASE
WORLD Server
144
140
142
Network

FIGURE 1

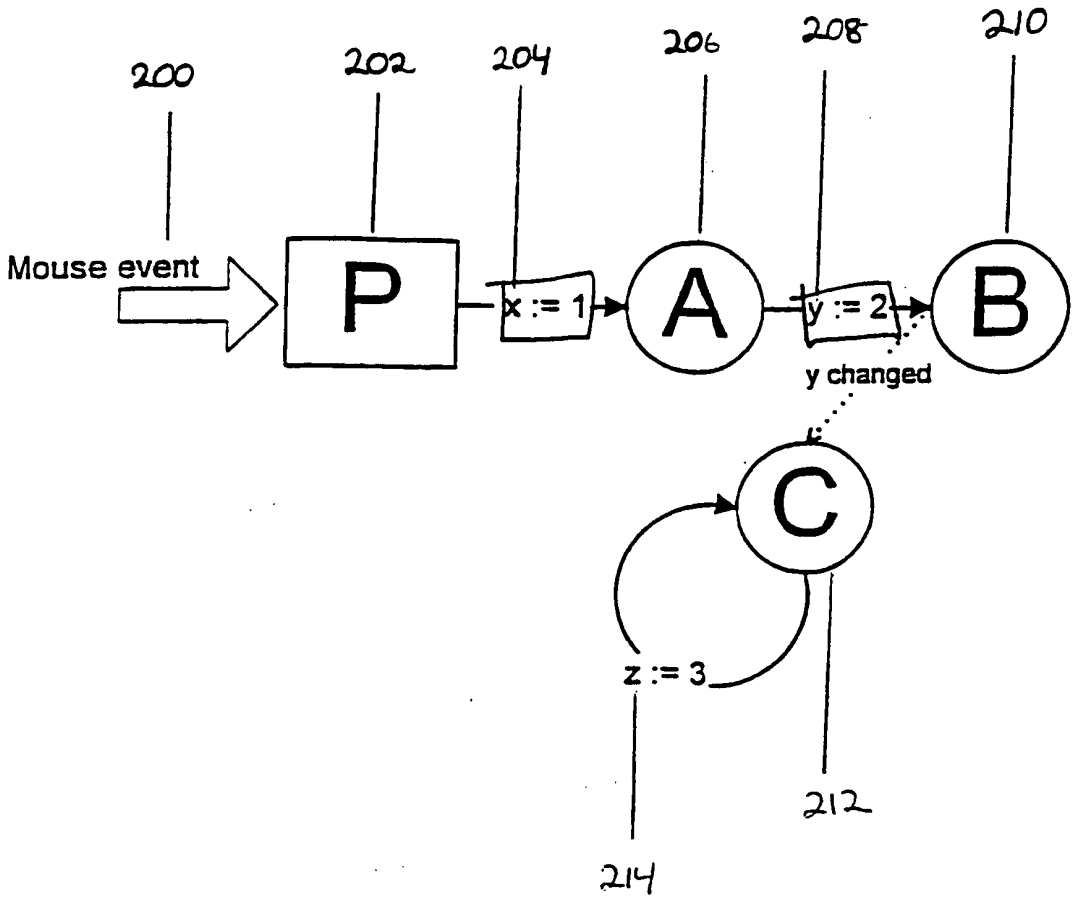

200
202
204
206
208
210
Mouse event
P
x := 1
A
y := 2
B
y changed
C
z := 3
212
214

FIGURE 2

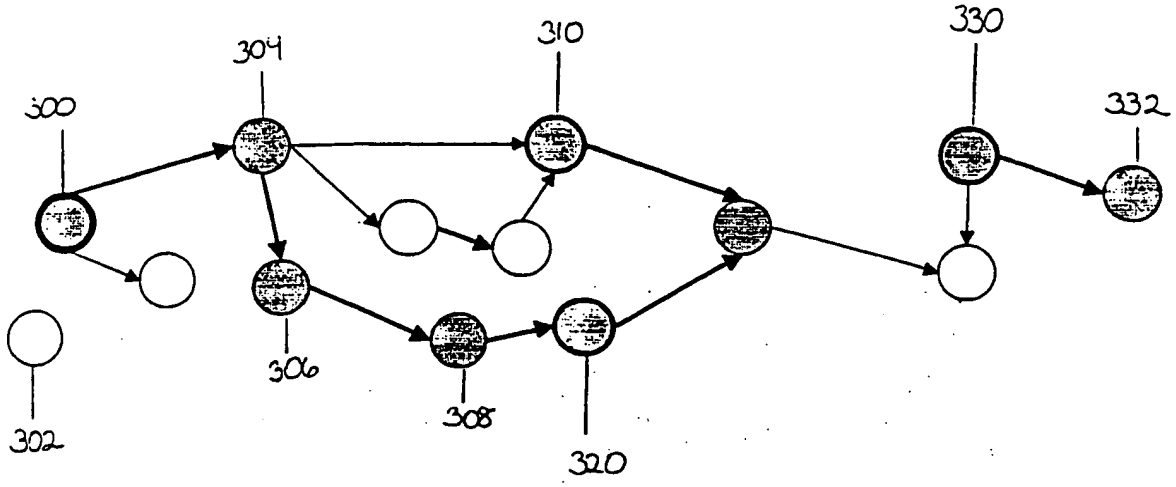


FIGURE 3(a)

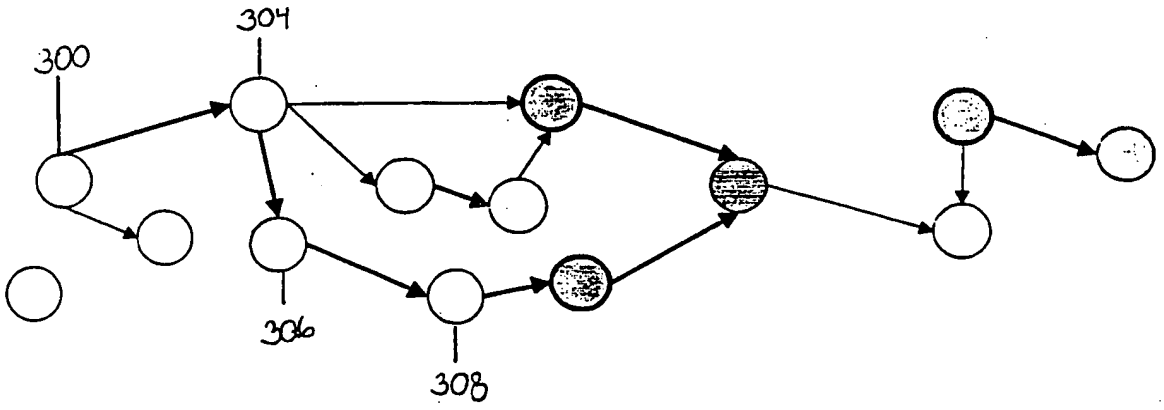


FIGURE 3(b)

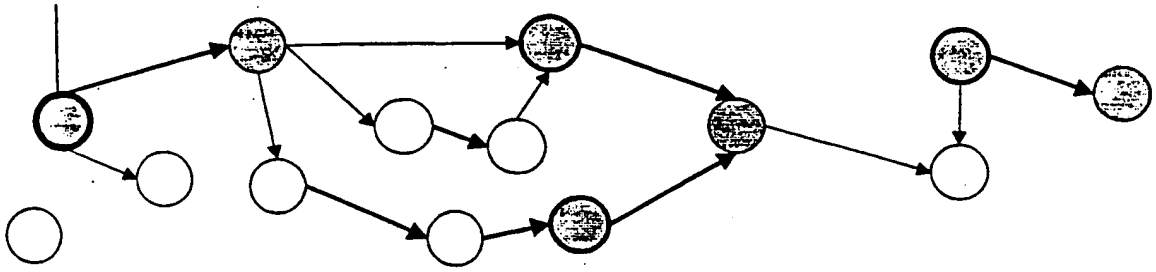


FIGURE 3(c)

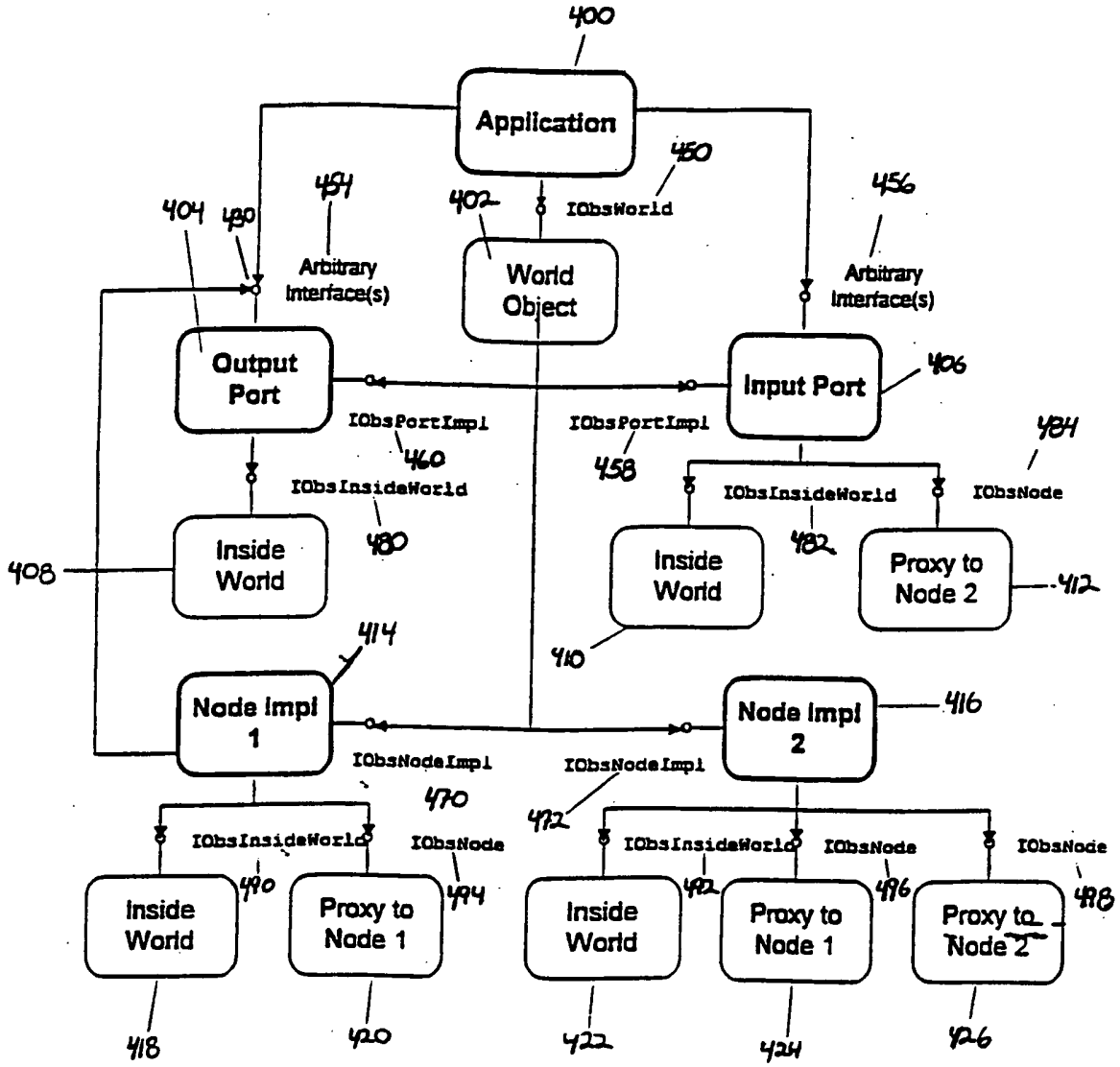

400
Application
450
IObsWorld
402
World Object
454
Arbitrary Interface(s)
456
Arbitrary Interface(s)
404
430
Output Port
Input Port
406
IObsPortImpl
460
IObsPortImpl
458
IObsInsideWorld
480
IObsInsideWorld
482
IObsNode
484
408
Inside World
Inside World
410
Proxy to Node 2
412
414
Node Impl 1
Node Impl 2
416
IObsNodeImpl
470
IObsNodeImpl
472
IObsInsideWorld
490
IObsNode
494
IObsInsideWorld
492
IObsNode
496
IObsNode
498
Inside World
Proxy to Node 1
Inside World
Proxy to Node 1
Proxy to Node 2
418
420
422
424
426

FIGURE 4

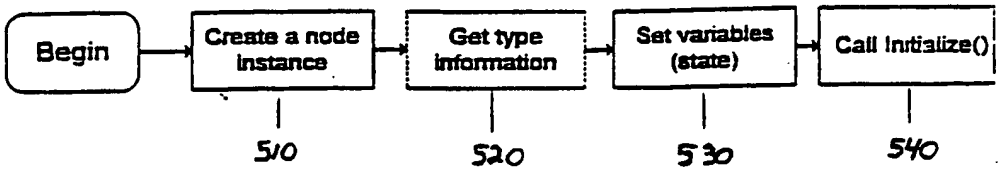

Begin
Create a node instance
Get type information
Set variables (state)
Call Initialize()
510
520
530
540

FIGURE 5

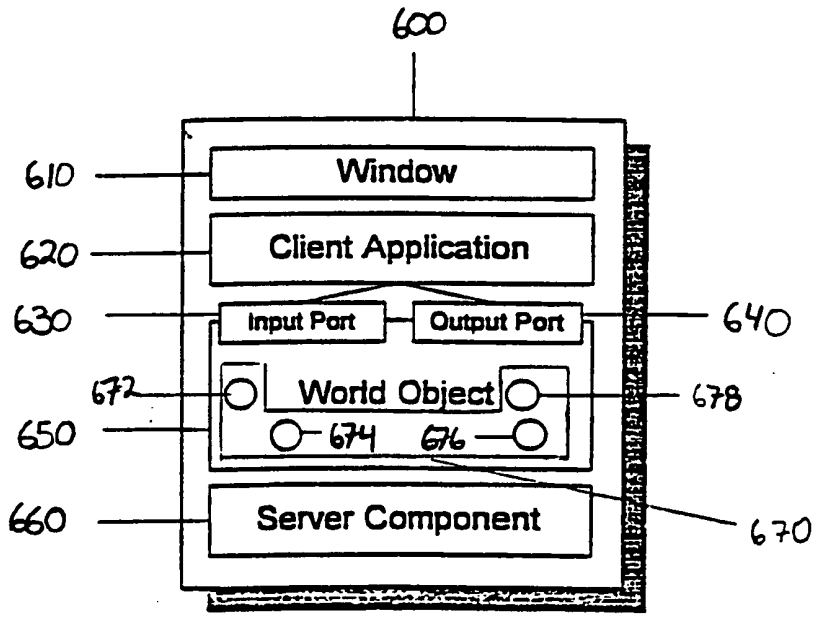

600
610
Window
620
Client Application
630
Input Port
Output Port
640
672
World Object
678
650
674
676
660
Server Component
670

FIGURE 6

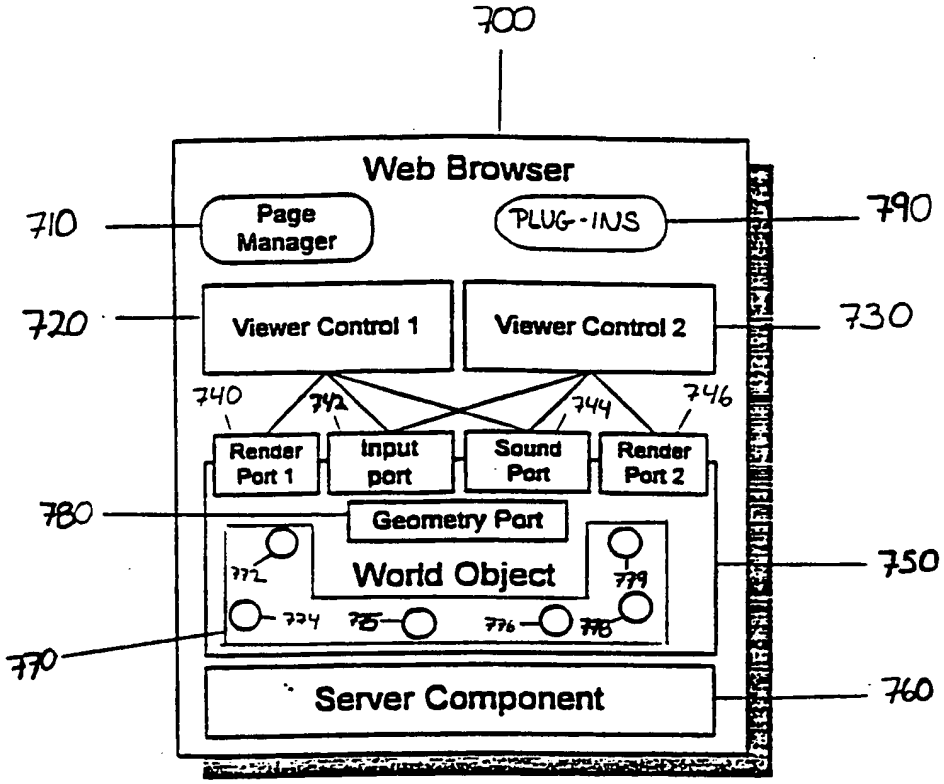

700
Web Browser
710
Page Manager
PLUG-INS
790
720
Viewer Control 1
Viewer Control 2
730
740
742
744
746
Render Port 1
Input port
Sound Port
Render Port 2
780
Geometry Port
772
World Object
779
774
775
776
778
750
770
Server Component
760

FIGURE 7

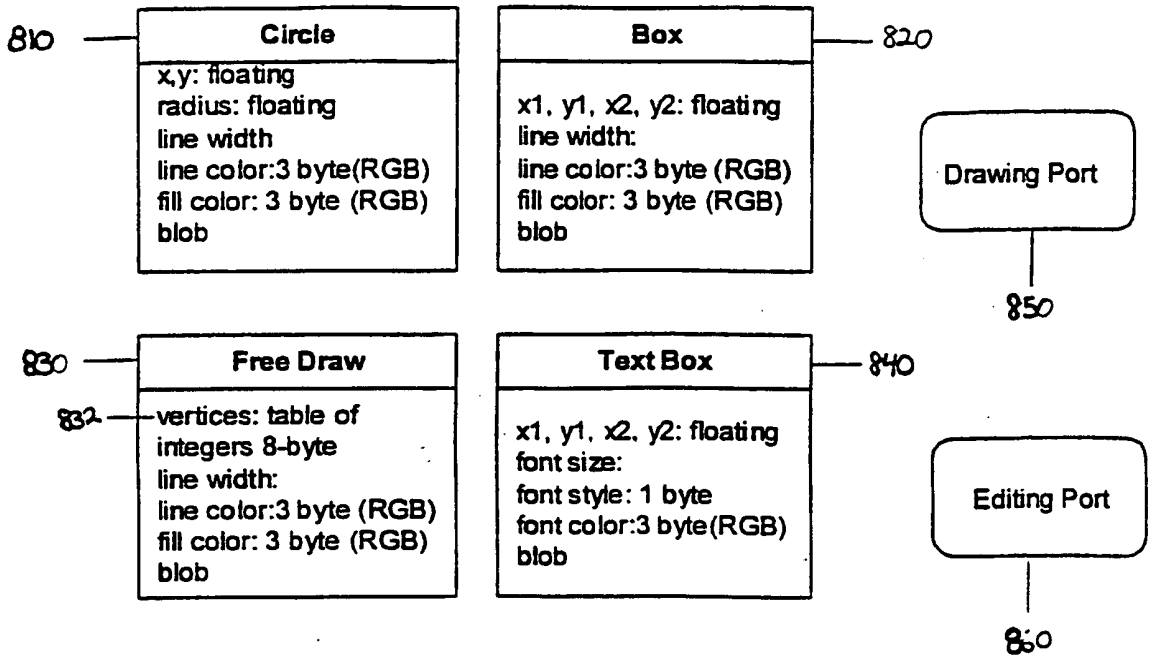

810
Circle
x,y: floating
radius: floating
line width
line color:3 byte(RGB)
fill color: 3 byte (RGB)
blob
Box
820
x1, y1, x2, y2: floating
line width:
line color:3 byte (RGB)
fill color: 3 byte (RGB)
blob
Drawing Port
850
830
Free Draw
832
vertices: table of integers 8-byte
line width:
line color:3 byte (RGB)
fill color: 3 byte (RGB)
blob
Text Box
840
x1, y1, x2, y2: floating
font size:
font style: 1 byte
font color:3 byte(RGB)
blob
Editing Port
860

FIGURE 8

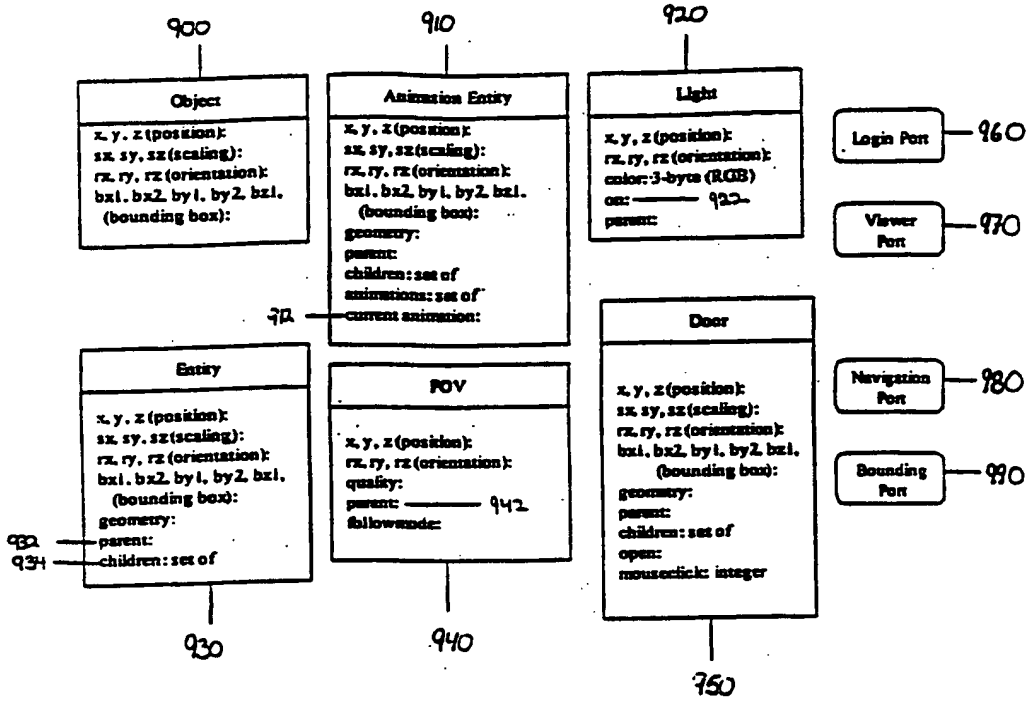

900
910
920
Object
x, y, z (position):
sx, sy, sz (scaling):
rx, ry, rz (orientation):
bx1, bx2, by1, by2, bz1,
(bounding box):
Animation Entity
x, y, z (position):
sx, sy, sz (scaling):
rx, ry, rz (orientation):
bx1, bx2, by1, by2, bz1,
(bounding box):
geometry:
parent:
children: set of
animations: set of
912
current animation:
Light
x, y, z (position):
rx, ry, rz (orientation):
color: 3-byte (RGB)
on:
922
parent:
Login Port
960
Viewer Port
970
Door
x, y, z (position):
sx, sy, sz (scaling):
rx, ry, rz (orientation):
bx1, bx2, by1, by2, bz1,
(bounding box):
geometry:
parent:
children: set of
open:
mouseclick: integer
Navigation Port
980
Bounding Port
990
Entity
x, y, z (position):
sx, sy, sz (scaling):
rx, ry, rz (orientation):
bx1, bx2, by1, by2, bz1,
(bounding box):
geometry:
932
parent:
934
children: set of
POV
x, y, z (position):
rx, ry, rz (orientation):
quality:
parent:
942
followmode:
930
940
950

FIGURE 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/21791

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) :G06F 13/00
US CL :709/219, 709/217, 709/204, 709/203; 707/201
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 709/219, 709/217, 709/204, 709/203; 707/201

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

west

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 6,058,397 A (BARRUS et al) 02 May 2000	1-16
A	US 5,889,951 A (LOMBARDI) 30 March 1999	1-16
A	US 5,841,980 A (WATERS et al) 24 November 1998	1-16
A	US 5,784,570 A (FUNKHOUSER) 21 July 1998	1-16
A	US 5,812,134 A (POOSER et al) 22 September 1998	1-16
A, P	US 6,020,885 A (HONDA) 01 Febuary 2000	1-16

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search
22 OCTOBER 2000

Date of mailing of the international search report
28 DEC 2000

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Authorized officer
GLENTON BURGESS

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☒ FADED TEXT OR DRAWING
- ☒ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☒ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.